

Cyber Assured Systems Engineering at Scale

Darren Cofer, Isaac Amundson, Junaid Babar, David Hardin, Konrad Slind
Collins Aerospace

Perry Alexander
University of Kansas

John Hatcliff, Robby
Kansas State University

Gerwin Klein
Proofcraft and University of New South Wales, Sydney

Corey Lewis
University of New South Wales, Sydney

Eric Mercer
Brigham Young University

John Shackleton
Adventium Labs

Abstract—Formal methods tools that provide mathematical proof of system properties have improved dramatically in their power and capabilities. Our team has developed a model-based systems engineering environment that integrates formal methods at all levels of system design. Our methodology and tools enable systems engineers to address cybersecurity concerns early in the development of complex high-assurance systems.

■ **AEROSPACE SYSTEMS ENGINEERS** are currently given few development tools to understand and mitigate potential cybersecurity vulnerabilities. Typically, they rely on process-oriented checklists and guidelines. Cyber vulnerabilities are often discovered during penetration testing late in the development process. Worse yet, they may be discovered only after the product has been fielded, necessitating extremely expensive and time-consuming remediation. This is not a sustainable development model.

Fortunately, formal methods tools have advanced to the point that they can be used to address cybersecurity and cyber-resiliency design challenges on real high-assurance systems at in-

dustrial scale, and do so much earlier in the development cycle. Our application domain is avionics and aerospace systems in general. This domain features large, real-time cyber-physical systems with the added complexities of performing safety-critical tasks as well as being exposed to a wide variety of cyber threats. Furthermore, aerospace systems are subject to intense regulatory scrutiny due to the certification requirements of this domain.

In previous work on the High-Assurance Cyber Military Systems (HACMS) project [1] we demonstrated that formal methods could be used to dramatically improve the cyber-resiliency of real aircraft, including an unmanned military helicopter. Our current work is focused on automating

the capabilities that we prototyped in the HACMS project and extending the reach and scale of the formal methods design and verification approach.

To this end, we have developed a model-based systems engineering (MBSE) environment that allows engineers to address a range of properties and manage system complexity through compositional analysis, integrating formal methods at all levels of the design process. MBSE processes utilize models as the primary vehicle for communication among the parties tasked with designing the system and as the primary design artifacts for requirements, verification, and code generation.

Our tools are based on the Architecture Analysis and Design Language (AADL) and extend the Open Source AADL Tool Environment (OS-ATE) [2]. The tools are specifically designed to bridge the gap between a user-level modeling language accessible to systems engineers and the highly specialized, formally verified code that implements the operating system (OS) kernel and other high-assurance components.

By using these tools to build real avionics systems, we show that current formal methods tools are practical, effective, and scalable to significant high-assurance applications in the aerospace industry.

INNOVATIONS

As part of the Cyber Assured Systems Engineering (CASE) project, our team has developed a MBSE tool environment that integrates design, verification, and code generation activities, enabling systems engineers to design-in cyber-resiliency for complex cyber-physical systems. The *BriefCASE* tools capture our vision for how formal methods can be applied throughout the design and build process to create high-assurance cyber-resilient systems.

A fundamental aspect of our approach is the use of architecture models to provide a framework for analyzing system behavior and organizing the assurance evidence produced. AADL allows engineers to describe the important elements of distributed, real-time, embedded systems (processors, memory, buses, processes, threads, and data interconnections) with sufficiently rigorous semantics that can support formal reasoning.

Proofs about models have less value without some way to ensure that the implementation

retains the properties of the model. The seL4 microkernel [3], used in both HACMS and CASE, is formally verified from its high-level security properties down to its binary implementation. By targeting seL4 we ensure that system components cannot interact in unintended ways and the *data flows* in the architecture model are enforced in the final product.

The main innovations of the BriefCASE tools and methodology are:

- 1) We provide automated architectural design patterns to address cyber-resiliency requirements, including synthesis of high-assurance components from formal specifications.
- 2) Our MBSE environment can target different operating systems including the seL4 microkernel, making its formal security guarantees easily accessible to developers. This ensures that the implementation produced is faithful to the modeled system.
- 3) Our approach is based upon co-evolution of system design and assurance artifacts, so that design changes automatically update the associated certification evidence.
- 4) Formal methods are integrated throughout the workflow, including requirements capture, component synthesis, verification, code generation, and the seL4 microkernel itself.

EXAMPLE

The example in Fig. 1 shows an AADL model of an unmanned air vehicle (UAV) for surveillance that was built with our BriefCASE tools. We will use this example to explain how the tools work together to implement the system and ensure its cyber-resiliency.

The system includes a ground station computer and the aircraft, consisting of a mission computer and a flight control computer. The baseline (unhardened) mission computer included only four of the software components shown in Fig. 1: the radio for communication with the ground station (*Radio*), the mission planning service (*Planner*, provided as legacy software running on Linux), flight plan waypoint segmentation (*WaypointManager*), and a serial interface to communicate with the flight control computer

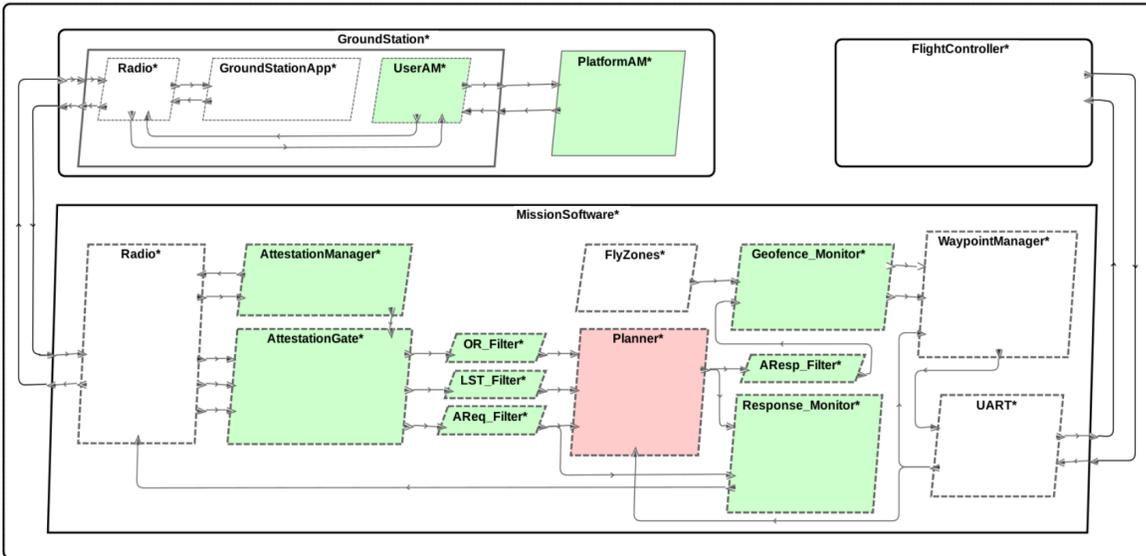


Figure 1. Cyber-resilient software architecture for UAV surveillance system with new high-assurance components (green) and virtual machine hosting legacy software (red).

(UART).

Cyber-threat analysis tools are used to analyze the unhardened functional model of the system, identifying the ground station and the mission planning service as primary sources of cyber attacks. Seven new cyber requirements are introduced that address ground station trust, message integrity, and run-time behavior vulnerabilities. The existing behavioral contracts in the unhardened system are strengthened to reflect these new requirements.

Design engineers use BriefCASE to transform the baseline system model to that shown in Fig. 1. Automated model transformations address the cyber requirements by inserting new high-assurance components (shown in green) and targeting the seL4 kernel to enforce separation between components. The *AttestationManager* establishes the trustworthiness of ground stations while the *AttestationGate* only passes messages from trusted sources. The three filters (*OR_Filter*, *LST_Filter*, and *AReq_Filter*) only pass well-formed messages received from the Radio. Another filter (*AResp_Filter*) on the output of the Planner ensures that only well-formed flight plans are sent to the *WaypointManager*. Two run-time monitors alert the system to suspicious behaviors from the Planner such as flight plans that enter *keep-out* zones or leave *keep-in*

zones (*Geofence_Monitor*), or unresponsiveness (*Response_Monitor*). The interface behavior of these high-assurance components, with the exception of the *AttestationManager*, is specified with assume-guarantee contracts (e.g., a filter makes no assumptions on input and only passes inputs that are syntactically well-formed).

The model is further transformed to move the mission planning service into a Linux virtual machine hosted on the seL4 microkernel. This permits us to run the legacy mission planner code without modification or porting and isolate any unintended behaviors. The target platform requires a static real-time schedule that is provided in the model. A transformation on the assume-guarantee contracts incorporates that schedule into the model for verification of the cyber requirements.

Finally, cyber-resilient mission computer software is automatically generated from the verified AADL model for execution by a fully verified version of seL4 running on an ODROID-XU4 (ARM Cortex-A7 CPU). We successfully demonstrated the resilience of the system against a variety of cyber attacks.

BRIEFCASE WORKFLOW

The BriefCASE environment provides systems engineers with a workflow and tool sup-

port for developing products with inherent cyber-resiliency.

The workflow starts with the development of a baseline AADL model of the system architecture focusing on the desired functionality. This model can be analyzed using any of the existing AADL tools (e.g., resource usage, information flow, latency) to determine whether it is acceptable. BriefCASE integrates additional tools that analyze the architecture model for cybersecurity vulnerabilities and generate requirements that, when addressed, will mitigate those vulnerabilities. These requirements are imported into the model and may be addressed using a collection of automated model transforms. As requirements are addressed in the design, an assurance case is updated with corresponding evidence, computed directly from the model or by supporting analysis tools. Code implementing new high-assurance components as well as communication and execution infrastructure is generated from the model along with associated assurance evidence.

The BriefCASE tools and their interactions are shown in Fig. 2. The following sections describe each step of the workflow in more detail, corresponding to the tools and artifacts shown in green in the figure.

Requirements Analysis

BriefCASE provides access to two analysis tools (GearCASE [4] and DCRYPPS [5]) that can examine AADL models to detect potential cyber vulnerabilities and suggest requirements for mitigation. Systems engineers are presented with a requirements management interface (top pane in Figure 3) for viewing the generated requirements and importing them into the model so they can be addressed. The interface enables engineers to select the requirements they wish to import and assign them unique identifiers, or omit them with rationale. A document listing the omitted requirements and rationale is maintained and may be a required development artifact for some certification domains.

Some requirements can also be formalized as assume-guarantee contracts added to the AADL model, enabling formal verification. Such a requirement will be imported into the model with with an associated formal contract.

A BriefCASE project contains a repository for

requirements. Imported requirements are represented as assurance case goals to be satisfied. For example, one of the requirements for well-formed command messages from the ground station (selected in Fig. 3) is imported as the following goal:

```
goal Req_WellFormed_OperatingRegion(comp_context : component) <=
  ** "UxAS component shall only receive well-formed messages" **
context Generated_On : "January 29, 2021";
context Req_Component : "MC::MissionComputer.Impl.SW.UxAS";
undeveloped
```

The goal is marked *undeveloped* initially. Evidential statements are added to the goal as the design is updated to address this requirement.

Cyber Transforms

To address the new cyber-resiliency requirement, the architecture will need to be transformed in such a way as to harden the design against the vulnerability. BriefCASE provides an extendable library of model transformations for addressing common cyber vulnerabilities. Currently, the following transformations are supported:

- Filter – Blocks messages that do not conform to the given specification
- Monitor – Detects violations of a given runtime condition and generates an alert
- Switch – Used with a Monitor to block messages when an alert is generated (also referred to as a *gate*)
- Attestation – Performs a measurement on non-local software to assess its trustworthiness
- Virtualization – Isolates software component(s) in a virtual machine
- Proxy – Inserts a pair of components to allow inspection of HTTPS message payloads
- seL4 – Transforms the model to comply with seL4 component properties

The transformations are automated by the BriefCASE tool, resulting in a hardened model that is correct-by-construction. For example, the requirement that a component only receives well-formed messages can be satisfied by the insertion of a high-assurance filter. A BriefCASE transform wizard helps to configure the filter component properties, including the filter behavioral specification, which is represented as an assume-guarantee contract.

BriefCASE then inserts a new filter component into the model, sets the component properties, and establishes the appropriate connections to source and destination components. The filter

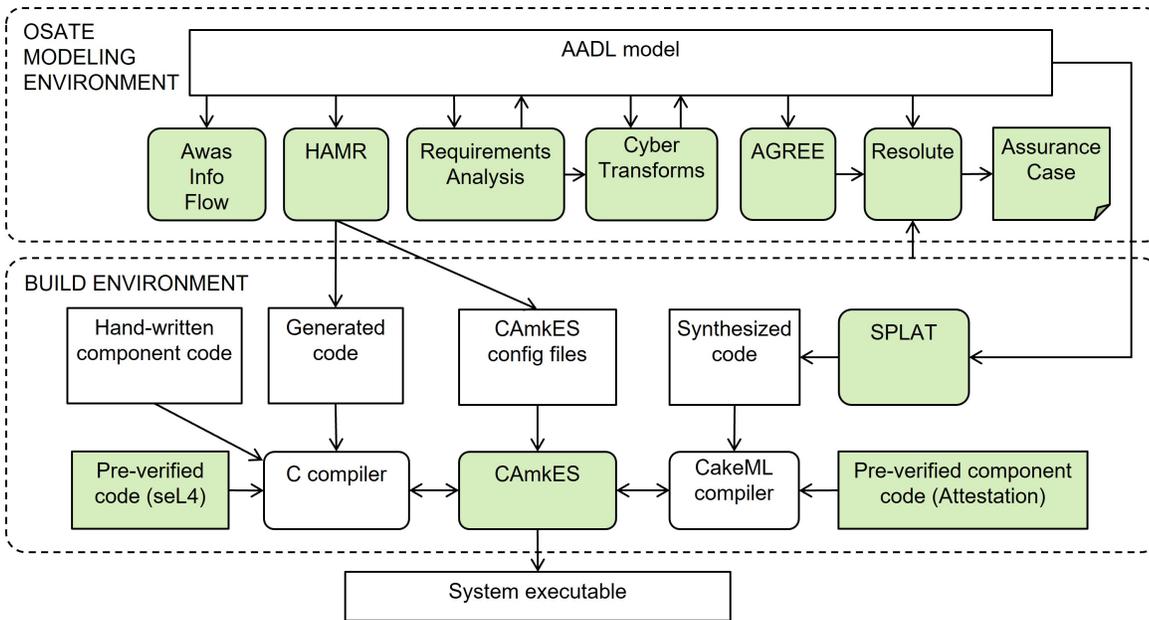


Figure 2. BriefCASE tool architecture. Tools and code/artifacts discussed in the BriefCASE workflow are shown in green.

behavioral contract is also added to the model, enabling formal analysis of the model as well as providing the behavioral specification for a provably correct synthesis of the filter component implementation.

The transformation also updates the assurance case with new evidential statements indicating that the associated goal has been satisfied, including the strategy used and links to context and associated evidence.

Compositional Analysis

The Assume Guarantee Reasoning Environment (AGREE) is a compositional, assume-guarantee-style model checker for AADL models [6]. AGREE attempts to prove properties about one layer of the architecture using properties allocated to subcomponents. The composition is performed in terms of assumptions and guarantees that are provided for each component. Assumptions describe the expectations the component has on the environment, while guarantees describe bounds on the behavior of the component.

AGREE uses k-induction as the underlying algorithm for model checking. AADL models and AGREE contracts are first translated into the Lustre language [7], including verification

conditions for consistency and correctness of the contracts. The model checker then attempts to find any model execution traces that would violate these conditions using one of several Satisfiability Modulo Theories (SMT) solvers. If the model checker covers all reachable states in the model without finding a violation then the properties are proved.

Once the system architecture has been modeled using AADL and the component behavior is specified using AGREE’s assume-guarantee contracts, we use the AGREE model checker to verify the consistency of these contracts.

- 1) Component interfaces – The output guarantees of each component must be strong enough to satisfy the input assumptions of downstream components.
- 2) Correctness of implementations – The input assumptions of a system along with the output guarantees of its *sub*-components must be strong enough to satisfy its output guarantees.

For example, in Fig. 1, the input assumptions of the Waypoint Manager must be weaker than the output guarantees of the Geofence monitor *and* the output guarantees of the Mission Soft-

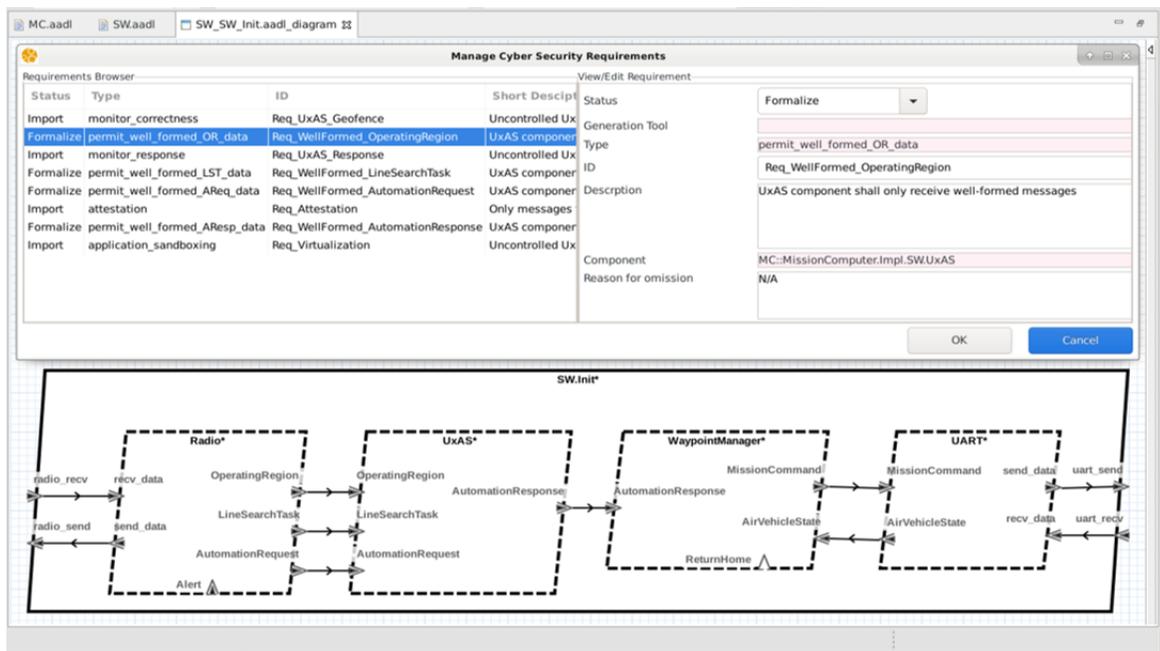


Figure 3. BriefCASE requirements management interface.

ware must be inferrable from its input assumptions combined with the output guarantees of its components.

This hierarchical strategy for reasoning about contracts, or *compositional analysis*, reduces the computational complexity of model checking by breaking down the larger problem into more manageable fragments.

High Assurance Component Synthesis

The correctness of the high-assurance components inserted by BriefCASE transformations means that each such component must meet its AGREE contract. This obligation is addressed by *formal synthesis*, using the Semantic Properties of Language and Automata Theory (SPLAT) tool. SPLAT generates code to implement the AGREE contract and then proves that its implementation exactly preserves the meaning of the contract all the way down to the binary for the target platform [8].

SPLAT uses the HOL4 theorem proving system to implement the synthesis and prove its correctness relative to the contract. The synthesis targets a dialect of Standard ML called *CakeML* and uses *CakeML*'s fully verified compiler to render the final binary [9]. *CakeML* is not a

real-time language since it is garbage collected, and although that has yet to be an issue in our use cases, it should be considered in applications where strict predictability is required.

The proof from SPLAT shows equivalence between the contract and the synthesized *CakeML* and then leverages the existing *CakeML* compiler proof for the rest. A final step of the proof reasons about the perpetual re-execution of the code as scheduled in a real-time environment. The modeling, relevant properties, and proof of that step is discussed in [10].

Remote Attestation

Semantic remote attestation is a technique for establishing trust in software running on a non-local computer. An appraiser requests an attestation from a target, receives evidence in response to the request, and appraises the evidence to determine trust [11]. Because remote attestation does not require modification of its measurement target, we utilize it to establish trust in legacy software that cannot otherwise be verified. We construct a verified remote attestation infrastructure around the legacy target that generates runtime and boot-time evidence.

In the UAV example, we wish to ensure that

the aircraft will only accept commands from trustworthy (non-compromised) ground stations. Our approach adds three attestation managers to the seL4-based architecture in Fig. 1. Each attestation manager executes protocols specified by Copland [12] phrases. Copland is a formal specification language designed for writing attestation protocols that are both verifiable and executable. The attestation managers themselves are written using Coq to verify attestation protocols and protocol interpreters, then synthesizing models to CakeML.

The *AttestationManager* on the Mission Computer makes attestation requests and appraises the results. A protocol request and nonce are sent to the remote target (in this case, the Ground Station), evidence is returned, and results appraised to determine trust. If the appraisal is successful, the identifier for the target is added to a list of trusted computers whose messages will be allowed to pass through the *Attestation-Gate*. Attestation can be requested periodically if warranted by the anticipated threats against the remote system.

Two other attestation managers must be added on the Ground Station. The Ground Station software has been modified to run in a Linux virtual machine hosted on seL4. The UserAM runs as a Linux process on this virtual machine. Its responsibilities include responding to attestation requests from the UAV, measuring the Ground Station application software, and requesting attestations from the Ground Station platform (the Linux kernel). When the UserAM receives an attestation request it responds by executing an attestation protocol that measures the application, requests measurements from the PlatformAM, signs the result, and generates a nonce from the request. The resulting evidence package is returned to the requesting appraiser on the UAV.

Because the UserAM runs as a Linux process, it cannot be trusted *a priori*. A PlatformAM is introduced to perform an attestation of the Linux kernel (and the UserAM). The PlatformAM runs as a seL4 component outside of the Linux virtual machine separated from the Linux kernel. seL4's guaranteed separation properties provide assurance that the PlatformAM cannot be compromised by other platform software. The PlatformAM only responds to requests from the

UserAM and similarly runs a protocol that produces signed results.

Information Flow Analysis

As systems become more complex, it is essential to have trustworthy methods to provide a common understanding of dependencies in the system and the respective responsibilities of the developers who may be from separate organizations. The Awas [13] AADL information flow analyzer and visualizer has been applied to enable developers and auditors to understand, reason about, explore, and visualize system dependencies and information flows at scale across components and subsystems. Awas processes the AADL system architecture model, specifically its inter-component connection descriptions and intra-component flow specifications, to provide formal system-wide impact and flow analyses. Such flows include component data/control flows, security-oriented information flows, and fault/error propagation specified using the AADL Error Modeling Annex (EMv2). Awas also provides a user-friendly Domain Specific Language (DSL) to query, check, and visualize custom safety/security system properties.

In our UAV example, Awas can compute and visualize how the information in Ground Station messages flows through the system as well as the components or ports that may directly or indirectly consume data derived from that information. Awas supports a number of forms of forward and backward interactive information queries. Using the Awas script-based query language, one can specify and check more complex properties, e.g., that information must flow through specific ports or components. These end-to-end flow specifications are often useful for supporting verification of the effectiveness of cyber-resiliency components. An Awas specification can state that information from untrusted components such as the Ground Station always flows through the Attestation Gate and filter components before reaching the Flight Control or other components that make critical decisions about the flight or mission of the UAV.

Real-Time Scheduling

In mixed-criticality mission systems, execution threads often have strict confidentiality, in-

tegrity, and availability requirements. Under certain conditions, timing channels can be manipulated to violate these security requirements. For example, temporal interference between timing channels can reduce availability of time-critical functionality and weaken the integrity of controls by inducing selective jitter. As a result, a compromised component could dominate the processor, preventing other components from completing their tasks, placing the security of the entire platform in jeopardy. Simple scheduling approaches that attempt to use priority schemes to mitigate this impact only protect the highest priority threads and leave the other threads vulnerable.

Temporal isolation is a more secure technique to restrict timing channels and reduce temporal interference between software threads executing on the same platform. BriefCASE achieves temporal isolation by leveraging prior work from safety and security-critical disciplines, such as avionics, where temporal isolation in real-time scheduling has been deployed for decades. We implemented a static cyclic scheduling approach using the seL4 domain scheduler, where a fixed schedule defines an ordered sequence of static execution slots. Each slot has a duration and a partition identifier. seL4 guarantees the start and end times for each temporal domain slice, according to the compile-time static cyclic schedule. This precisely constrains when each thread may execute, backed by seL4 verification proofs. System designers are responsible for the determining the temporal requirements, such as worst case execution times (e.g., by test or analysis) and latency bounds, of each thread, based on mission requirements. These requirements may be captured in the AADL model, and offline tools are available to generate the static cyclic schedule based on the model. Determining the mission temporal requirements is not part of the BriefCASE tools.

BriefCASE generates a start-of-frame synchronization signal for each thread using a special thread called the Pacer, which sends periodic signals to each thread. Each application thread blocks until it receives its signal from the Pacer. The application thread runs to completion, and then blocks again on the Pacer signal for the next iteration. Each thread subsequently executes exactly once during its statically scheduled time

slice.

The new seL4 Mixed-Criticality Systems (MCS) variant provides additional capability that can support temporally isolated real-time systems. As part of BriefCASE we developed a proof-of-concept static cyclic scheduler for MCS. It includes a start-of-frame signal, which eliminates the need for the Pacer component. It also includes kernel level support for flexible dynamic scheduling that satisfies some real-time properties, such as period and execution time.

Infrastructure Code Generation

The High Assurance Modeling and Rapid engineering for embedded systems tool (HAMR) [14] is a multi-platform, multi-language AADL code generation framework. In the CASE project, HAMR is primarily used to generate code for deployment for the seL4 microkernel, but system and component prototyping is also supported utilizing HAMR's code generation capability for the Java Virtual Machine (JVM) and Linux.

Using seL4 as a foundation, HAMR enables AADL to be used as a model-based development and systems engineering framework for seL4-based applications. One of the primary objectives is to support system builds that leverage seL4 separation and information flow guarantees to achieve the AADL-specified component isolation and inter-component communication needed for cyber-resiliency. HAMR ensures that seL4 is configured to permit the exact inter-component information flows analyzed and visualized by Awas at the model level.

For each AADL thread component, HAMR generates a thread code skeleton and application programming interfaces (APIs) for communicating over the ports declared on the component. For components that are implemented manually, the developer fills out the thread skeleton with application code. HAMR supports coding component application logic in either C, Slang [15] (a high-assurance subset of Scala that can be translated to C), or CakeML.

HAMR generates component infrastructure and integration code implementing the semantics of AADL-compliant thread scheduling, thread dispatching, and port-based communication. For port communication, shared memory communication (AADL data ports), buffered messaging

(AADL event data ports), and buffered notification (AADL event ports) are supported. HAMR code generation is staged using a translation architecture that facilitates adding new backends for different target platforms. Almost all the infrastructure code is implemented in Slang, which can then be used for JVM deployments or translated to C for Linux or seL4 deployments. The Slang-based implementation of the AADL runtime framework can be viewed as a high-level reference implementation of AADL semantics. Automatic translation of this reference implementation to C on different platforms helps establish semantic consistency across those platforms.

The seL4 deployment uses the component architecture for microkernel-based embedded systems (CAMkES) code-generation framework to configure the microkernel. The HAMR generated CAMkES directly encodes the AADL model's component and communication topology and includes the AADL run-time infrastructure with its thread scheduling. HAMR leverages the existing seL4 domain scheduler to enforce time partitioning and provide static cyclic scheduling.

HAMR also supports Linux-based virtual machine components in the seL4 deployment and the ability to run the entire system on the QEMU emulator. HAMR automatically configures virtual machine based components, and this feature is used to sandbox the untrusted legacy code for the Mission Planner in the example UAV system. The QEMU emulator support facilitates rapid prototyping for test, debug, and analysis, and it enables automated regression testing.

As part of its code generation process, HAMR produces flow graphs reflecting the inter-component information flow at both the AADL architecture level and the CAMkES level for the seL4 deployment. Visual representations are provided for manual inspection, and SMT-based representations are generated for formal reasoning. The SMT-based representations are used to prove the following properties:

- 1) All AADL modeled flows are in the CAMkES configuration.
- 2) No extraneous flows have been added to the CAMkES configuration.

These properties focus on cyber-resiliency, but other key semantic properties can be verified as

well.

Secure Microkernel

The seL4 microkernel [3] is a lightweight, fast, and secure operating system kernel. Its implementation is fully formally verified, from high-level security properties down to the binary level. It was the first OS kernel with this degree of formal verification, and after more than a decade of further research and engineering is still not only the leading formally verified OS kernel, but also the fastest OS kernel on the Arm architecture.

Its formal verification makes seL4 the ideal basis for high-assurance systems. It is in itself a demonstration of the level of fidelity and scale formal verification can achieve [16]. seL4 supports multiple architectures (Arm, x86-64, RISC-V), provides deep security properties such as integrity, confidentiality and availability, and comes with formally verified user-level system initialization. As one of its multiple available OS personalities, seL4 also offers the user-level CAMkES component system that provably achieves isolation between statically specified components.

The formal proofs about seL4 and the corresponding user-level components measure over one million lines of proof script in total. They constitute one of the largest continually maintained formal proof artifacts in existence and provide a rich target for new techniques in proof engineering, proof repair, and automation for constructing new proofs about software as well as maintaining existing large-scale proof artifacts.

While it is essential to build a high-assurance system on a high-assurance OS kernel, this is not the main feature seL4 provides for systems engineering — a simpler real-time OS might be formally verified, but would not be sufficient for the engineering method described in this paper. The true power of seL4 lies in its ability to scale formal analysis and verification to the much larger code bases that make up entire systems. It does so by providing strong isolation between user-level components [17].

This isolation means that components can be analyzed separately from each other and be composed safely — in this way seL4 provides the foundation that the soundness of the highly automated analysis tools such as AGREE depend on. It makes it possible to run entire untrusted

virtual machines and securely monitor their behavior on the same hardware. It makes it possible to provide filter and monitor components and prove that these components cannot be tampered with by the components they protect. And it makes it possible to guarantee that the limited communication channels that the analysis tools assume to exist in the AADL model are the *only* communication channels that are available to the components in the system. The combination of these enables automated high-level analysis with high assurance.

Assurance Case

Assurance activities for high-integrity systems in the aerospace domain are currently driven by industry and government standards such as DO-178C and MIL-HDBK-516C. The use of assurance cases to show conformance to standards (or provide alternate means of assurance) is being pursued in separate research programs. In addition, we have found that in assuring the cyber-resiliency properties of aircraft designs we need to integrate different kinds of evidence with varying levels of formality. This has been our motivation for incorporating assurance case methods in BriefCASE.

In previous work, we developed the *Resolute* language and tool [18] as a way to help engineers create an assurance argument describing the steps taken during the design process to make the system safe and secure. The *Resolute* syntax supports construction of assurance cases that comply with the Goal Structuring Notation (GSN) v2 standard. Claims are expressed as *goals* and *strategies*, and can contain attributes such as *context*, *assumptions*, and *justification*. Claims can be marked *undeveloped*, which *Resolute* interprets as an unsupported claim, or with a *solution*, which is an explicit assertion that the claim is supported. Rather than being a separate document, a *Resolute* assurance case is part of the architecture model and can refer to elements within the model. Since it is not a static representation, it can ensure that the assurance argument remains consistent with the evolving design.

A partial assurance case for the hardened UAV surveillance system generated by *Resolute* is shown in Fig. 4. The assurance case includes subtrees corresponding to the high-assurance com-

ponents used to satisfy cyber-resiliency requirements and a subtree associated with HAMR code generation and traceability to seL4 separation guarantees.

BriefCASE includes a library of *Resolute* assurance strategies, or *patterns*, that align with the CASE workflow. The patterns are instantiated with context from the AADL model and specify the evidence required to support the cyber-resiliency goals of the system. For example, the `add_filter` strategy is automatically inserted into the assurance case when the *Filter* transformation is performed, and includes logical rules that *Resolute* uses to determine whether the claim of well-formed messages is supported by evidence. The `add_filter` definition includes the following sub-goals (shown in Fig. 4):

- Component property implemented – The filter has been implemented correctly to meet its AGREE specification, as shown by the proof produced by SPLAT.
- Filter is connected – The filter component is still present at the correct location in the model and has not been altered or deleted by subsequent design changes.
- Filter cannot be bypassed – There is no alternate information flow in the model that would allow the filter to be bypassed and therefore not perform its function.
- AGREE properties are valid – The filter specification has been verified by AGREE to meet its intended purpose in the system.

AIRCRAFT APPLICATION

We have successfully demonstrated our BriefCASE methodology and tools to develop proof-of-concept enhancements for the Common Avionics Architecture System (CAAS), an integrated cockpit avionics suite developed by Collins Aerospace. CAAS serves as a prime example of modern air platform complexity with common avionics across a variety of Army, Navy, and Air Force platforms including the CH-47F, CH-53K, MH-60, and Mission Enhanced Little Bird (MELB). The CAAS application offered an opportunity to apply BriefCASE tools across a variety of mission systems, including legacy components, flight critical software, as well as new and evolving systems.

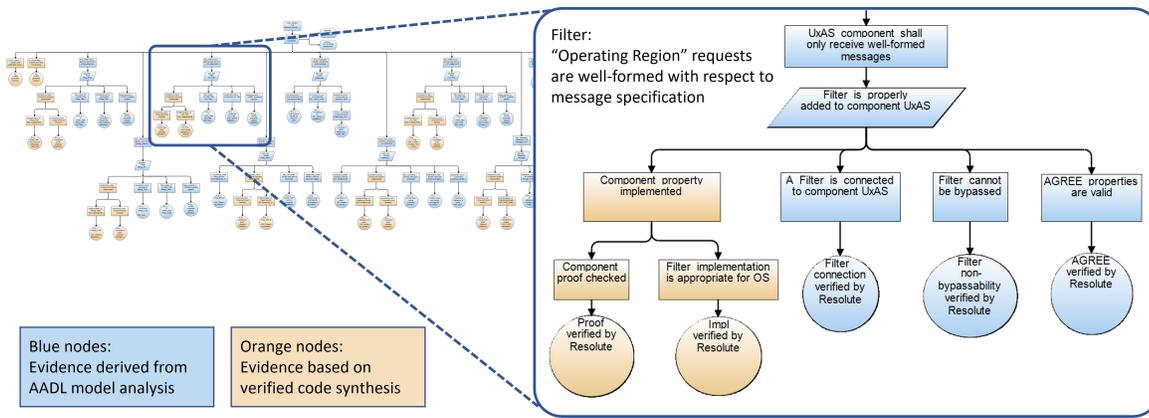


Figure 4. Resolute assurance case for hardened UAV surveillance system (partial).

Most of the work for this demonstration was performed by a team of CAAS development engineers who had no previous experience with formal methods. Our primary goal was to enable CAAS product engineers to employ the CASE tools to create an operational mission scenario exhibiting cyber threats and mitigations that could be exercised using the facilities of the Collins CH-47F System Integration Laboratory.

The CH-47F demonstration system for CASE focused on integrating pilot and soldier wireless tablet computers for increased situational awareness and display of Automatic Dependent Surveillance-Broadcast (ADS-B) data regarding nearby air traffic (Fig. 5). BriefCASE tools were used to implement this networking enhancement while ensuring that no new cybersecurity vulnerabilities were introduced. A high-assurance gateway was added between the existing CAAS network and the new wireless network, including new components for monitoring messages to and from the wireless devices. Remote attestation was also added to ensure that any devices that attempt to join the wireless network are running trustworthy software. This also required configuring the seL4 microkernel to run on an existing CAAS processing module (the video processing module, or VPM) that was repurposed to serve as the secure gateway.

The CAAS engineers first developed an AADL model of the CH-47F CAAS system. They added the enhanced capabilities for wireless access described above, resulting in a base-

line architecture. The engineers then analyzed their baseline AADL architecture utilizing the GearCASE and DCRYPPS tools, resulting in a set of cyber requirements. They employed the BriefCASE AADL tools to add filters, monitors, attestation gates, and seL4 isolation to satisfy the cyber requirements. ADS-B anti-spoofing monitors were also developed by the CAAS team to detect inconsistencies in aircraft position and velocity trends, bad traffic identifiers, and other possible indicators of spoofed aircraft.

The specifications for the filters and monitors were developed by the CAAS engineers using the AGREE contract language, with assistance from our research team. The SPLAT tool was used to synthesize the monitors and filters from the AGREE specifications with high assurance, as described above. The HAMR tool was then used to generate infrastructure code for the overall system running on seL4.

The tablet operating environment was modified to run application software in a Linux virtual machine hosted on seL4. This was done so that the remote attestation component for platform measurement could be securely isolated from the (untrusted) application software.

An additional complication was the need to add proxy and network adapter components to allow the encrypted internet messages from devices on the untrusted wireless network to be examined by the secure gateway. Dedicated processing cores were available on the VPM for the provisioning of these “low-side” and “high-

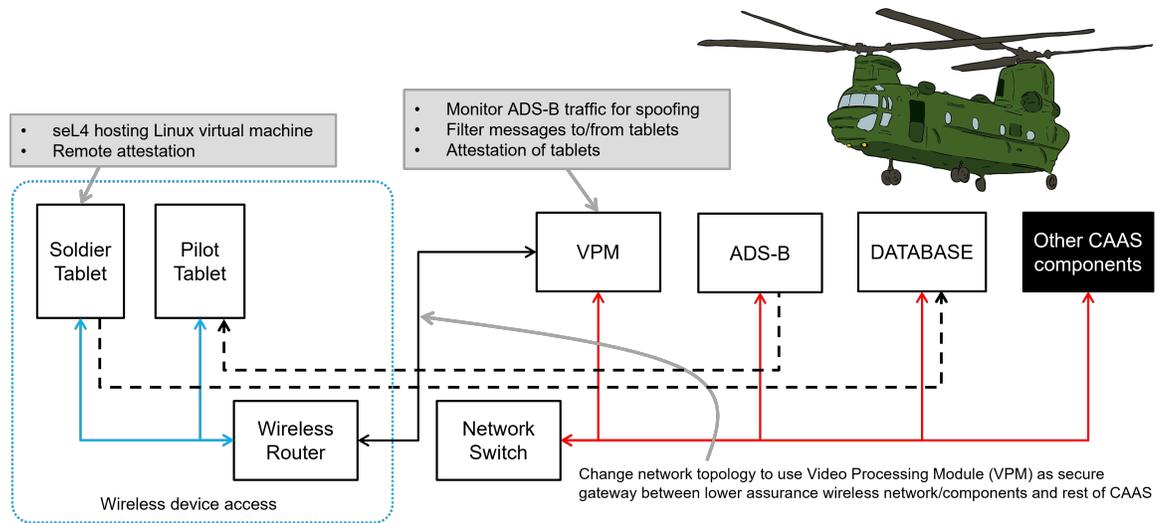


Figure 5. Cyber-resilient software architecture adding wireless device access to CH-47 avionics.

side” components separately from the other high-assurance software. The architecture model was transformed to add these components but their implementation code had to be manually written due to the complexity of dealing with off-the-shelf networking technologies. Automating the synthesis of network adapter and proxy components is future work.

The commercial board used for development and the Collins hardware equivalent in the helicopter demonstration system (both ARM Cortex-A53 CPUs) use the same seL4 code base as the Fig. 1 example. However, the seL4 proofs have not been updated for these hardware platforms and so they have not been fully verified. We expect that there would be only minor differences from the existing proofs for similar platforms.

As initial users of the BriefCASE tools, the CAAS engineers encountered some limitations in the specification expressiveness and documentation, as might be expected for the first use of research tools by product area developers. This led to improvements in the tools to add or extend capabilities. Initial estimates of processing time required for the complex monitoring components turned out to be optimistic, requiring an optimization effort. Lack of detailed documentation and hardware instabilities hampered the tablet software development effort. But overall the BriefCASE tools were able to be productively used by

Collins product engineers to produce the CH-47F CAAS demonstration system on time and within budget, providing our research team with valuable feedback on the strengths and weaknesses of the current BriefCASE tool environment.

CONCLUSION

We have produced a model-based systems engineering environment called BriefCASE, based on the Architecture Analysis and Design Language. The BriefCASE tools and methodology provide design, analysis, and code generation capabilities based on formal methods and targeted at high-assurance cyberphysical systems.

Key innovations include automated architectural design patterns for cyber-resiliency, co-evolution of system design and assurance artifacts (captured as an assurance case linked to the architecture model), synthesis of code for high-assurance components, and code generation targeting the formally verified seL4 microkernel.

We have demonstrated the effectiveness and scalability of these tools by using them to add new cyber-resilient features to a military helicopter avionics system. Their successful use by our product engineers provides evidence that formal methods can be incorporated into industrial projects.

All of the tools are open-source, with links and documentation available at <http://loonwerks>.

com/projects/case.html. We hope that others will find value in this approach and extend the tools with new cyber transforms, expanded system analysis tools, and code generation for additional operating systems and computing platforms.

ACKNOWLEDGMENT

This work was funded by DARPA contract HR00111890001 and has been approved for public release, distribution unlimited. The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

REFERENCES

1. D. D. Cofer, A. Gacek, J. Backes, M. W. Whalen, L. Pike, A. Foltzer, M. Podhradsky, G. Klein, I. Kuz, J. Andronick, G. Heiser, and D. Stuart, "A formal approach to constructing secure air vehicle software," *Computer*, vol. 51, no. 11, pp. 14–23, 2018.
2. P. H. Feiler and D. P. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*, 1st ed. Addison-Wesley Professional, 2012.
3. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: formal verification of an OS kernel," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSOP 2009, Big Sky, Montana, USA, October 11-14, 2009*, J. N. Matthews and T. E. Anderson, Eds. ACM, 2009, pp. 207–220. [Online]. Available: <https://doi.org/10.1145/1629575.1629596>
4. T. Patten, D. Mitchell, and C. Call, "Cyber attack grammars for risk-cost analysis," in *Proceedings of the 15th International Conference on Cyber Warfare and Security*, Norfolk, VA, 2020.
5. R. Laddaga, P. Robertson, H. E. Shrobe, D. Cerys, P. Manghwani, and P. Meijer, "Deriving cyber-security requirements for cyber physical systems," *CoRR*, vol. abs/1901.01867, 2019. [Online]. Available: <http://arxiv.org/abs/1901.01867>
6. D. Cofer, A. Gacek, S. Miller, M. W. Whalen, B. LaValley, and L. Sha, "Compositional verification of architectural models," in *NASA Formal Methods*, A. E. Goodloe and S. Person, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 126–140.
7. P. Caspi, D. Pilaud, N. Halbwegs, and J. A. Plaice, "LUSTRE: A declarative language for programming synchronous systems," in *In 14th Symposium on Principles of Programming Languages (POPL'87)*. ACM, 1987.
8. E. Mercer, K. Slind, I. Amundson, D. Cofer, J. Babar, and D. Hardin, "Synthesizing verified components for cyber assured systems engineering," in *24th International Conference on Model-Driven Engineering Languages and Systems (MODELS 2021)*, October 2021.
9. R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, "CakeML: a verified implementation of ML," in *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, S. Jagannathan and P. Sewell, Eds. ACM, 2014, pp. 179–192. [Online]. Available: <https://doi.org/10.1145/2535838.2535841>
10. J. Å. Pohjola, H. Rostedt, and M. O. Myreen, "Characteristic formulae for liveness properties of non-terminating CakeML programs," in *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*, ser. LIPIcs, J. Harrison, J. O'Leary, and A. Tolmach, Eds., vol. 141. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 32:1–32:19. [Online]. Available: <https://doi.org/10.4230/LIPIcs.ITP.2019.32>
11. G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen, "Principles of remote attestation," *International Journal of Information Security*, vol. 10, no. 2, pp. 63–81, June 2011.
12. J. Ramsdell, P. D. Rowe, P. Alexander, S. Helble, P. Loscocco, J. A. Pendergrass, and A. Petz, "Orchestrating layered attestations," in *Principles of Security and Trust (POST'19)*, Prague, Czech Republic, April 8-11 2019.
13. H. Thiagarajan, J. Hatcliff, and Robby, "Awas: AADL information flow and error propagation analysis framework," *Innovations in Systems and Software Engineering (ISSE)*, 2021.
14. J. Hatcliff, J. Belt, Robby, and T. Carpenter, "HAMR: An AADL multi-platform code generation toolset," in *10th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, ser. LNCS, vol. 13036, 2021, pp. 274–295.
15. Robby and J. Hatcliff, "Slang: The Sireum programming language," in *10th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, ser. LNCS, vol. 13036, 2021, pp. 253–273.
16. G. Klein, J. Andronick, K. Elphinstone, T. C. Murray,

- T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an OS microkernel," *ACM Trans. Comput. Syst.*, vol. 32, no. 1, pp. 2:1–2:70, 2014. [Online]. Available: <https://doi.org/10.1145/2560537>
17. G. Klein, J. Andronick, M. Fernandez, I. Kuz, T. C. Murray, and G. Heiser, "Formally verified software in the real world," *Commun. ACM*, vol. 61, no. 10, pp. 68–77, 2018. [Online]. Available: <https://doi.org/10.1145/3230627>
 18. I. Amundson and D. Cofer, "Resolute assurance arguments for cyber assured systems engineering," in *Design Automation for Cyber-Physical Systems and Internet of Things (DESTION 2021)*, May 2021.

Darren Cofer is a Fellow at Collins Aerospace. His research interests include formal methods and tools for verification and certification of high-integrity systems. He received his PhD in Electrical and Computer Engineering from the University of Texas at Austin and is a senior member of the IEEE. Contact him at cofer@ieee.org.

Isaac Amundson is a Senior Principal Research Engineer at Collins Aerospace. His research focuses on modeling, formal verification and assurance, and he has led the design and implementation of multiple MBSE toolchains for providing inherent dependability in high-assurance systems. He received his PhD in Computer Science from Vanderbilt University.

Junaid Babar (PhD, Iowa State University) is a research engineer at Collins Aerospace. His research interests include formal verification via model checking, temporal logics and decision diagrams.

David Hardin is a research engineer at Collins Aerospace, and has made contributions in the areas of formal methods, high-assurance computer architecture, and real-time embedded Java. He is editor of the book *Design and Verification of Microprocessor Systems for High-Assurance Applications*. He received a PhD in Electrical and Computer Engineering from Kansas State University.

Konrad Slind (PhD, TU Munich) is an industrial logician at Collins Aerospace. His research interests include design and implementation of higher order logic theorem provers, verified compilation, and self-describing data formats.

Perry Alexander is the AT&T Foundation Distinguished Professor and Director of The Information and Telecommunication Technology Center at The

University of Kansas. His research interests include trusted computing, remote attestation, formal verification and formal synthesis. He received his PhD from The University of Kansas.

John Hatcliff is a University Distinguished Professor and Lucas-Rathbone Professor of Engineering in the Computer Science Department at Kansas State University. His research interests include software architecture, formal verification, and interoperability in the avionics, automotive, and medical domains. He received his PhD in Computer Science from Kansas State University.

Robby is a Professor of Computer Science at Kansas State University working in the area of formal methods, software engineering, and programming languages. He received a NASA Turning Goals Into Reality award in 2003, a NSF CAREER award in 2007, an ICSE 2000 Most Influential Paper award in 2010, and an ACM SIGSOFT Impact award in 2010.

Gerwin Klein is Chief Scientist at Proofcraft and Conjoint Professor at UNSW Sydney. His research interests include software verification, semantics of programming languages, and proof engineering. He is the architect of the seL4 microkernel verification which received the SIGOPS Hall of Fame Award. He received his PhD in Computer Science from TU Munich.

Corey Lewis is a Senior Proof Engineer at UNSW Sydney. He has been involved in all stages of the verification of the seL4 microkernel.

Eric Mercer is an Associate Professor at Brigham Young University. His research interests include software verification, model checking, static analysis, and programming languages. He received his PhD in Electrical Engineering from the University of Utah.

John Shackleton is a Senior Principal Research Scientist at Adventium Labs, focused on real-time embedded systems, cybersecurity, and model-based system engineering and analysis.